Electronic Communications of the EASST
Volume 18 (2009)

Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)

Parallelization of Graph Transformation Based on Incremental Pattern Matching

Gábor Bergmann, István Ráth, Dániel Varró

15 pages

Guest Editors: Artur Boronat, Reiko Heckel
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Parallelization of Graph Transformation Based on Incremental Pattern Matching

Gábor Bergmann[1], **István Ráth**[2], **Dániel Varró**[3]

[1 2 3] (bergmann,rath,varro)@mit.bme.hu, http://www.inf.mit.bme.hu/FTSRG/
Méréstechnika és Információs Rendszerek Tanszék (MIT),
Budapesti Műszaki és Gazdaságtudományi Egyetem (BME), Budapest, Hungary

Abstract: Graph transformation based on incremental pattern matching explicitly stores all occurrences of patterns (left-hand side of rules) and updates this result cache upon model changes. This allows instantaneous pattern queries at the expense of costlier model manipulation and higher memory consumption.

Up to now, this incremental approach has considered only sequential execution despite the inherently distributed structure of the underlying match caching mechanism. The paper explores various possibilities of parallelizing graph transformation to harness the power of modern multi-core, multi-processor computing environments: (i) incremental pattern matching enables the concurrent execution of model manipulation and pattern matching; moreover, (ii) pattern matching itself can be parallelized along caches.

Keywords: graph transformation, incremental pattern matching, parallelization

1 Introduction

Nowadays, a main challenge of software engineering is the adaptation to parallel computing architectures. In order to increase execution speed, algorithm designers need to think of new ways to exploit the computing power of multi core processors instead of purely relying on more efficient processor designs. Experience has shown that this is a complicated task: whether parallel execution can actually be effectively applied depends largely on the problem itself.

Model transformation is an application domain where speed optimization based on parallel execution has a lot of potential, especially in case of large, industrial models. In fact, model transformations seem to be an ideal target for parallel execution as in practical transformations, many similar, or almost identical model structures need to be traversed and transformed. Frequently, these model manipulation sequences are non-conflicting, which naturally calls for an execution model where these sequences are executed on the available processors in parallel.

Using a graph transformation (GT) [EEKR99] based approach for model transformations, there are even more possibilities for the exploitation of parallelism. Besides model manipulation sequences, graph transformations involve a graph searching phase, which is targeted at finding the matches of a graph pattern. However, despite the recent optimization activities in the graph transformation community, which have been reported at tool contests [SNZ08, Gra08], GT tools rarely exploit parallel execution for further improvement both in terms of execution speed and scalability with model sizes.

Incremental pattern matching [BÖR+08] offers an entirely different execution model compared to local search-based implementations. The match sets for all patterns involved in the graph transformation are computed in an initialization phase prior to execution (e.g. when the model itself is loaded into memory), and as the transformation progresses, this match set cache is incrementally updated as the model graph changes (update phases). Thus, model search phases are reduced to fast read-from-cache operations, in exchange for the overhead imposed by cache update phases which occur synchronously with model manipulation operations. Benchmarking [BHRV08] has shown that in certain scenarios, this approach leads to several orders-of-magnitude increases in speed.

In the current paper, we introduce novel extensions to the incremental pattern matcher of the VIATRA2 framework, which is based on RETE networks [For82], to exploit parallelism based on asynchronous model updates and multi-threaded match set caching.

First, *update phases may be executed concurrently* to the model transformation's main execution thread. In this case, the cache validation thread of the match set may execute concurrently with model manipulation sequences or textual output emission, e.g. in the case of code generation transformations. This approach aims to reduce the overhead imposed by update phases, in the case when parallel computing power is available.

Then, if further scaling up is required, the implementation of the *match set cache updating can be multi-threaded*. It is important to point out that both of these approaches are significantly different from parallelized pattern search. Finally, as incremental pattern matching provides fast cache-reading operations, it supports *parallel transformation execution* by allowing simultaneous access to caches from multiple threads. By improving this scenario with concurrent update phases, model manipulation protected by locks will no longer force other transformation threads to wait for the termination of the time-consuming update. As a consequence, read-intensive transformations are expected to scale well with parallel computational capacity.

The rest of the paper is structured as follows. Section 2 gives a brief introduction on graph transformations. Section 3 describes RETE, and its implementation in the VIATRA2 model transformation framework. The main contributions of the paper are presented in Section 4, where we present ways of parallelizing both pattern matching and model manipulation. Implementation details are revealed in Section 5. Finally, we discuss related work in Section 6 and conclude the paper in Section 7.

2 Foundations of model transformation

This section gives an overview on the foundations of the specification and simulation of modeling languages. In order to specify the abstract syntax of the modeling language, the concept of metamodeling is used. For transforming models to other models or generated code, and simulating the behaviour of models, the paradigm of graph transformation [Roz97] is applied.

2.1 Model transformation example: Petri nets

In this paper, we will use the transformation of Petri nets as a demonstration for parallelization concepts. These demonstrating Petri net transformations include Petri net firing as a model

simulation example.

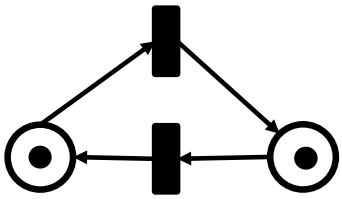


Figure 1: A sample Petri net.

Petri nets (Figure 1) are widely used to formally capture the dynamic semantics of concurrent systems due to their easy-to-understand visual notation and the wide range of available analysis tools. Petri nets are bipartite graphs, with two disjoint sets of nodes: *Places* and *Transitions*. Places may contain an arbitrary number of Tokens. A token distribution (marking) defines the state of the modelled system. The state of the net can be changed by firing enabled transitions. A transition is enabled if each of its input places contains at least one token and no place connected with an inhibitor arc contains a token (if no arc weights are considered). When firing a transition, we remove a token from all input places (connected to the transition by Input Arcs) and add a token to all output places (as defined by Output Arcs).

2.2 Foundations of metamodeling

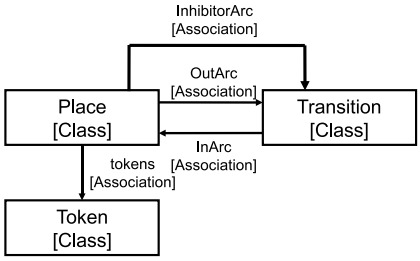


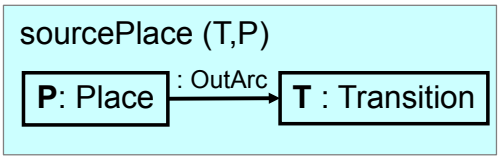
Figure 2: Petri net metamodel.

A *metamodel* describes the abstract syntax of a modeling language. Formally, it can be represented by a type graph. Nodes of the type graph are called *classes*. A class may have attributes that define some kind of properties of the specific class. *Inheritance* may be defined between classes, which means that the inherited class has all the properties its parent has, but it may further contain some extra *attributes*. *Associations* define connections between classes. Figure 2 shows a simple Petri net metamodel.

2.3 Graph patterns and graph transformation

Graph patterns are frequently considered as the atomic units of model transformations [VB07]. They represent conditions (or constraints) that have to be fulfilled by a part of the instance model in order to execute some manipulation steps on the model. A basic graph pattern consists of graph elements corresponding to the metamodel. A *negative application condition* (NAC) prescribes contextual conditions for the original pattern which are forbidden in order to find a successful match. Figure 10 presents a simple graph pattern consisting of a Place P, a Transition T and an OutArc A to enumerate the source places connected to a given transition.

Graph transformation (GT) [EEKR99] provides a high-level rule and pattern-based manipulation language for graph models. Graph transformation rules can be specified by using a left-hand side – LHS (or precondition) graph (pattern) determining the applicability of the rule, and a right-hand side – RHS (postcondition) graph (pattern) which declaratively specifies the result model after rule application. Elements that are present only in (the image of) the LHS are




```
pattern sourcePlace(T, P) = {
  transition(T);
  place(P);
  outArc(A, P, T);
}
```

Figure 3: Matcher for the *sourcePlace* pattern

deleted, elements that are present only in the RHS are created, and other model elements remain unchanged (in accordance with the single-pushout approach in VIATRA2). For instance, a GT rule may specify how to remove (or add) a token from a place, as shown in Figure 4.

```
// Removes a token from the place 'Place'.
gtrule removeToken(in Transition, in Place) = {
 precondition find sourcePlaceWithToken
    (Transition, Place, Token);
 postcondition find sourcePlaceWithoutToken
    (Transition, Place, Token);
}
// Adds a token from the place 'Place'.
gtrule addToken(in Transition, in Place) = {
 precondition find targetPlaceWithoutToken
    (Transition, Place);
 postcondition find targetPlaceWithToken
    (Transition, Place, Token);
}
```

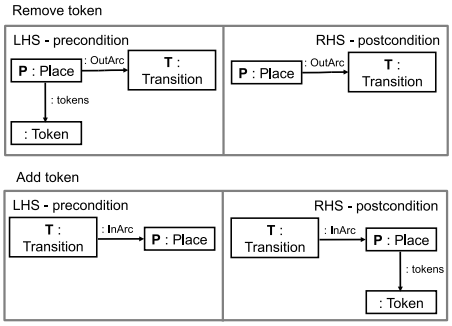


Figure 4: Graph transformation rules for firing a transition

Complex model transformation can be assembled from elementary graph patterns and graph transformation rules using some kind of control language. In our examples, we use abstract state machine (ASM) [BS03] for this purpose as available in the VIATRA2 framework. The following transformation simulates the firing of a transition, i.e. the removal of tokens from input places and the addition of tokens to output places (see Figure 5).

```
rule fireTransition(in T) = seq {
 if (find isTransitionFireable(T)) // confirm that the transition is fireable
 seq {
  forall Place with find sourcePlace(T, Place) // remove tokens from all source places
   do apply removeToken(T, Place); // GT rule invocation
  forall Place with find targetPlace(T, Place) // add tokens to all target places
   do apply addToken(T, Place);
 }
}
```

Figure 5: Transformation program for firing a transition

3 RETE-based incremental graph pattern matching

The incremental graph pattern matcher of the VIATRA2 framework adapts [BÖR+08] the RETE algorithm, which is a well-known technique in the field of rule-based systems.

RETE network for graph pattern matching RETE-based pattern matching relies on a network of nodes storing *partial matches* of a graph pattern. A partial match enumerates those model elements which satisfy a subset of the constraints described by the graph pattern. In a relational database analogy, each node stores a *view*. Matches of a pattern are readily available at any time, and they will be incrementally updated whenever model changes occur.

Input nodes serve as the underlying knowledge base representing a model. There is a separate input node for each entity type (class), containing a view representing all the instances that conform to the type. Similarly, there is an input node for each relation type, containing a view consisting of tuples with source and target in addition to the identifier of the edge instance.

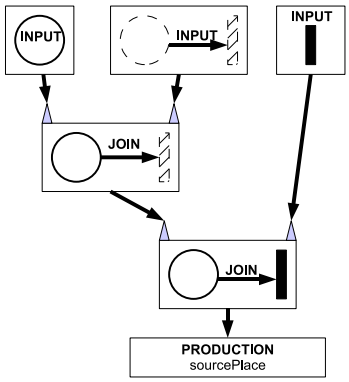


Figure 6: Simple RETE matcher

At each *intermediate node*, *set operations* (e.g. filtering, projection, join, etc.) can be executed on the match sets stored at input nodes to compute the match set which is stored at the intermediate node. The match set for the entire pattern can be retrieved from the output *production node*. One kind of intermediate node is the *join node*, which performs a natural join on its parent nodes in terms of relational algebra; whereas a *negative node* contains the set of tuples stored at the primary input which do *not* match any tuple from the secondary input (which corresponds to anti-joins in relational databases).

As an illustration, Figure 6 shows a RETE network matcher built for the *sourcePlace* (see Figure 10) pattern illustrating the use of join nodes. By joining three input nodes (the top-most nodes on Figure 6), this sample RETE net enforces two entity type constraints ('Place' and 'Transition' entity types on the left and right input nodes) and an edge (connectivity) constraint (corresponding to the relation connecting the 'Place' and 'Transition' entity types), to find pairs of Places and Transitions connected by an out-arc.

Updates after model changes. Input nodes receive notifications about each elementary model change (i.e. when a new model element is created or deleted) and release an update token on each of their outgoing edges. Such an update token represents changes in the partial matches stored by the RETE node. Positive update tokens reflect newly added tuples, and negative updates refer to tuples being removed from the set. Upon receiving an update token, a RETE node determines how the set of stored tuples will change, and release update tokens of its own to signal these changes to its child nodes. This way, the effects of an update will propagate through the network, eventually influencing the result sets stored in production nodes.

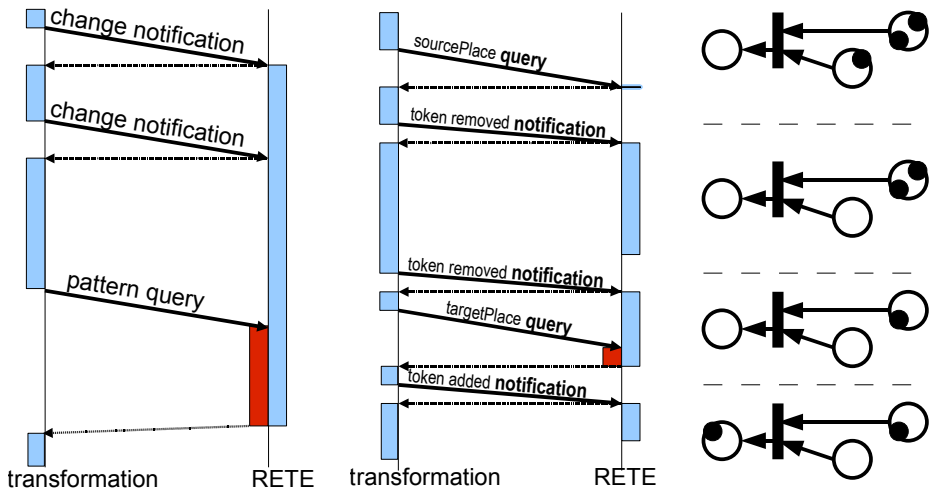
The match set can be retrieved from the network instantly without re-computation, which makes pattern matching very efficient. As a trade-off, there is increased memory consumption, and update operations become more complex.

4 Parallel transformation with incremental pattern matching

This section presents our conceptual contributions to the parallel execution of model transformations. First, Subsection 4.1 will discuss in detail how the asynchronous RETE approach allows the update phases to be executed in the background, while the transformation continues uninterrupted. In Subsection 4.2, we generalise this approach to multiple RETE threads for systems with more than two CPU cores, based on the multi-threaded RETE maching set cache. The proposed pattern matcher is applied to a multi-threaded model manipulation context in Subsection 4.3 to let the model manipulation phase take advantage of the number of CPU cores.

4.1 Concurrent pattern matching and model manipulation

Contrary to our previous work, the RETE net implementation used throughout this paper relies on *asynchronous* message passing. This involves a *message queue* attached to the network, containing update messages manifested as objects. Each message object specifies a recipient node, the tuple representing the update, and the sign (insertion or deletion). The message consumption cycle fetches the first message from the queue and delivers it to the appropriate node; the node will place any propagated output messages to the end of the queue, thereby achieving asynchronous messaging. Change notifications issued by model manipulation are simply put into the queue; then the update propagation phase consists of looping the message consumption cycle until the queue becomes empty.



(a) General concept (b) *fireTransition* concurrently (c) Petri net states

Figure 7: Concurrent pattern matching

Using asynchronous messaging, the load on the main thread of the transformation can be reduced by executing the incremental pattern matcher (which consumes change messages from the queue) in a separate thread. When the transformation manipulates the model (see Figure 7(a)), it only has to send the new update message to the message queue, and continue its operation.

The thread of the pattern matcher will execute the update propagation in the background, ideally, without imposing a performance penalty on the transformation thread. When the message queue becomes empty, the RETE network has reached a *fixpoint*; the pattern matcher thread then goes to sleep and will not resume its operation until a new update message is posted.

When the transformation initiates pattern matching, it has to assure that background update propagations have terminated and the matches stored at the production nodes are up-to-date; so if necessary, it will have to sleep until RETE reaches its fixpoint.

Figure 7(b) shows how the Petri net firing rule *fireTransition* (defined in Figure 5) may behave in such a concurrent system. (i) First, the set of source places is fetched instantaneously from the pattern matcher. (ii) Then, one token is deleted in every source place, each of them issueing a notification to the pattern matcher thread that results in some update propagation in the RETE net. (iii) Next, the list of target places is retrieved after update propagation is finished. (iv) Finally, a new token is created at each target place, resulting in subsequent notifications. Figure 7(c) displays the corresponding states of the Petri net model.

Initial performance results. While the local search based pattern matchers operate with cheap model changes and costly pattern queries, the sequential RETE-based matcher [BÖR+08] has a moderate overhead on model change balanced by instant pattern queries. This novel concurrent incremental pattern matching approach combines the advantages of the former two: it has cheap model manipulation costs, and potentially instant pattern queries. Although the transformation might have to wait for the termination of the background pattern matcher thread, the worst case of this time loss is still comparable to the update overhead of the original RETE approach.

This concurrent approach is expected to improve performance over a non-concurrent implementation (as described in Section 3) if there are comparatively infrequent pattern matcher queries and complex model changes between them. This would correspond to a *forall* style control flow when all matches of a pattern are obtained first, and then each of them is processed (potentially) simultaneously, which is common in model-to-model transformation scenarios. This complements the traditional advantage of incremental pattern matching, which manifested especially on *as long as possible* style control flows: when single matches are selected and processed one by one until there are no matches of the pattern.

Initial experiments[1] have shown that the concurrent approach improves performance by up to 20% on the Sierpinsky benchmark of [SNZ08]. For building a Sierpinsky-triangle of 8, 9 and 10 generations, our original RETE ran for 2.6s, 8.3s, and 26.2s, while the concurrent solution took 2.2s, 6.9s, 22.8s to terminate, respectively.

4.2 Multi-threaded pattern matching with RETE

The concurrent patten matching approach can be improved further given that the hardware architecture is capable of running multiple threads efficiently. There are various approaches of parallelizing the RETE algorithm, see Section 6 for details. Here we present a simple solution.

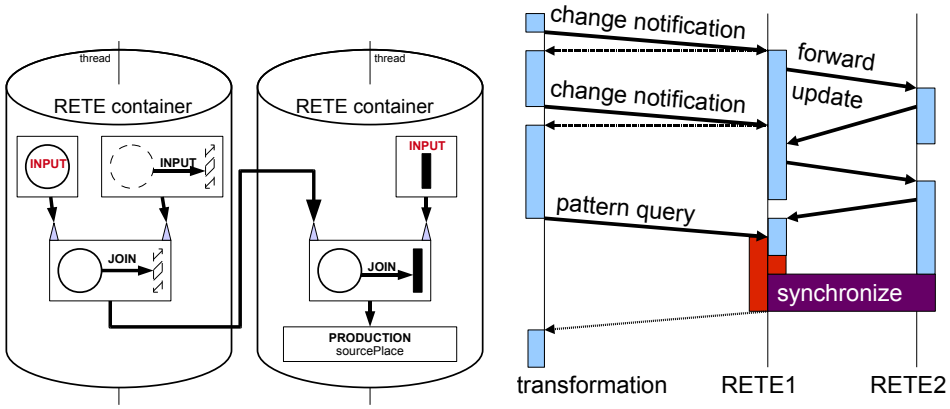
The basic idea is to employ multiple pattern matcher threads to consume update messages. However, if these threads share the same message queue and RETE nodes, and multiple threads

[1] Environment: 2.2GHz Intel Core 2 Duo processor, Windows Vista, Sun Java 1.6.0_11, 1GB heap memory

could access the same node simultaneously, this could easily lead to complex inconsistency problems, which could not be easily avoided by locks.

Our proposal splits the network into separate RETE *container*s, each of which is responsible for matching a set of subpatterns. A container has its own distinct set of nodes, and assigns each RETE container to a dedicated pattern matcher thread consuming update messages of a dedicated queue. Each container is responsible for forwarding messages to its nodes using the dedicated message queue. This way, two threads are not allowed to operate on the same RETE node, thus maintaining mutual exclusions is not necessary.

Forwarding messages between two containers is accomplished by enqueueing the message in the target container. Figure 8(a) depicts a parallel version version of Figure 6 illustrating how a RETE net can be split into several containers for parallel execution.



(a) RETE containers (b) Message sequence

Figure 8: Multi-threaded pattern matching

Figure 8(b) illustrates how changes performed by the transformation induce update propagation in the RETE net. The update propagation spreads between containers, and is processed in parallel. When the transformation needs to match a pattern, however, it will have to synchronize with the pattern matcher and wait until all RETE activity has settled.

If a container runs out of update messages to process, it reaches a *local fixpoint*, otherwise it remains *active*. The *global fixpoint* is reached when all containers are in a local fixpoint. In order to retrieve up-to-date and consistent match sets, the transformation thread has to wait for a global fixpoint. This thread synchronisation goal, however, is not trivial to accomplish, since a container can leave its local fixpoint and become active again before a global fixpoint is reached due to incoming messages from other, still active containers. To address this issue, we have developed and implemented a termination protocol based on logical clocks.

Termination protocol for RETE containers Each container C_i is equipped with a logical *clock* (denoted as $clock_i$) that is incremented whenever a local fixpoint is reached by the message consumption thread of the container (denoted $thread_i$). Each time container C_i sends an update message to container C_k, the message is appended to the message queue of C_k and the value of

$clock_k$ is retrieved and stored in c_i as $criterion_i[k]$, all as a single atomic step[2]. The retrieved clock value imposes a *termination criterion*: the network can only reach a global fixpoint if the value of $clock_k$ exceeds the received snapshot. This means that the relayed message has been delivered to the recipient node in C_k and all of the (local) consequences have been resolved, resulting in a new local fixpoint in C_k, which is required for a global fixpoint.

When C_i reaches its local fixpoint, it atomically increments its clock and reports the event to a *global RETE network object*; this report includes the incremented $clock_i$, along with the values $criterion_i[k]$ for each k. Similarly, when the transformation changes the model and consequently sends a change notification (formulated as an update message) to an input node in container C_k, it hands over the message to the message queue of C_k, fetches the $clock_k$ from that container and reports it to the network object as a termination criterion, also performed as a single atomic step.

The global network object maintains an array $criterion_{global}[k]$ storing the *largest* reported criterion for each k, and $clock_{reported}[i]$ for the latest clock value reported by C_i. Upon receiving the report, the global network object evaluates whether a global fixpoint is reached and wakes the transformation thread when appropriate. Determining whether a global fixpoint holds is as simple as checking, for each container, whether the highest reported termination criterion value stemming from that container is exceeded by its the latest reported fixpoint-time clock value. This Termination Condition is formulated as:

$$\forall_k : clock_{reported}[k] > criterion_{global}[k] \tag{1}$$

The above protocol is proven to be able to determine global fixpoints. The proof and further details are available in [Ber08].

Performance discussion. It is important to point out that the performance of such a system may depend highly on the amount of synchronization and replication that is necessary when messages are passed between the containers. In theory, it would seem beneficial if the subpatterns (deployed to separate RETE containers) had a low number of interconnections, but further research is necessitated to achieve this in practice. An ideal application scenario would be several transformations or parts of the same transformation that are known to use different patterns; allowing easy, straightforward splitting and parallelisation of the RETE net, with a low amount of inter-connectedness. By partitioning the patterns into relatively independent containers, a multi-threaded RETE pattern matcher may achieve high performance.

4.3 Multi-threaded model manipulation

In case of incremental pattern matching, the usefulness of optimizing the pattern matcher has its limitations, as significant CPU time is spent on the rule application itself. Further gains in performance can only be achieved by accelerating the execution of rule application and model manipulation. For this purpose, we exploit multi-core architectures to provide multi-threading for the model manipulation component as well.

[2] the outlined procedure is only necessary if $k \neq i$; messages sent and received within the same container can use the message queue without any interaction with clocks

Multi-threading for model manipulation can be achieved easier if pattern matching is performed in a separate thread, as described in Subsection 4.1 (or multiple threads, as in Subsection 4.2). When model manipulation threads change the model, they send update notifications atomically, which involves inserting an update message addressed to the appropriate input node into the message queue of the node's container. When a transformation thread requires the set of matches for a certain pattern, the pattern matcher call returns them immediately if the network is in a global fixpoint, or suspends the thread (but not others) until that fixpoint state is reached.

Conflict prevention Several approaches aim to achieve serializable (i.e. thread-safe) parallelisation of graph transformation rule applications, either for rule instances within one transformation sequence, or for separate transformation runs. Most advanced solutions exploit the declarative nature and concurrency theory of graph transformation (critical pairs, etc.) to execute non-conflicting rules in parallel [Mez07].

Unfortunately, in many practical cases, model transformations are intertwined with hard-to-analyze imperative actions, or simply, there are too many conflicts. Furthermore, even if conflict-free high-level behaviour is guaranteed, there can still be conflicts caused by low-level implementation details (concurrent access to edge lists, etc.) that have not been taken into account. Therefore low-level solutions are necessary for ensuring exclusion in parallel execution.

Locking scheme Exclusion can be provided by imposing a locking system on the model space. If a thread acquires a lock, other threads will have to wait until it is released before they can acquire a conflicting lock. The lock system can have, for instance, a model-level, an element-level, or hierarchy-based lock granularity; locks may be held for the length of an elementary model manipulation operation, or longer sequences; also, a read lock / write lock model is preferable. However, our experiments (see below) showed that great care must be taken in the choice of locking strategy.

A high ratio of lock conflicts could mean that the lock scheme itself becomes a bottleneck and prevents the improvement of performance by parallelization, therefore locks should be compatible whenever possible. On the other hand, over-specializing locks for the sake of compatibility would result in too many individual lock acquisitions, which can have a considerable overhead. Therefore the transformation system must strike a balance in order to achieve good scalability even in write-intensive scenarios. The locking strategy can be predetermined, configurable by the transformation author, or decided by reasoning on the declarative description of the graph rules. This is a challenge that necessitates further research.

Initial performance results. A high amount of synchronization can diminsh performance both through waiting and overhead. Parallel execution of read-intensive transformations, however, is relatively conflict-free, therefore it does not heavily suffer from waiting, and can scale up to multiple CPU cores efficiently. A suggested application scenario would be parallel code generation, with each thread producing a separate output file from a corresponding aspect of the source model. Code generation is usually a read-only operation; we also believe that using different aspects of the model aids in the partitioning of the RETE net.

Our current VIATRA2 implementation supports multiple transformation threads, concurrent

pattern matching, and model-level R/W locking. We used this system to measure the performance of parallel code generation, namely generating PNML [JKW02] descriptions of several Petri nets within the model space. This application scenario has the advantage that transformation jobs are entirely read-only. However, since all generation jobs basically follow the same algorithm and use similar Petri-nets, it does not easily lead to partitioning the RETE net; therefore, we used concurrent pattern matching, but with a single RETE container. The initial expriments[3] show that this system has a quasi-linear scalability. A Petri-net generated by the procedure in [BHRV08] as "Size 50000" was used as a sample input. Two PNML code generators in sequence ran for 2.9s each, 5.8s altogether. Two code generators in parallel ran for 3.7s each, but they took only 3.9s altogether.

On the other hand, we have also conducted preliminary measurements on the AntWorld case study of GraBaTs 2008 [Gra08]. Our parallel solution to this benchmark turned out not to scale well; in fact, it was slower than the sequential one. The results have showed that a naive locking scheme (the entire model has to be locked on each elementary operation) results in poor scalability in case of a write-intensive transformation like AntWorld.

Due to the complexity (and sometimes strange characteristics) of parallel algorithms, further measurements are required to compare its performance with non-incremental approaches.

5 Integration into the VIATRA2 model transformation framework

This section describes how the concepts described in Section 4 were incorporated into VIATRA2. Subsection 5.1 gives a basic overview of the architecture of VIATRA2 and how it was changed to support parallelized graph transformation. Subsection 5.2 briefly introduces how the proposed parallelization techniques are made available in the transformation language of VIATRA2.

5.1 Architecture

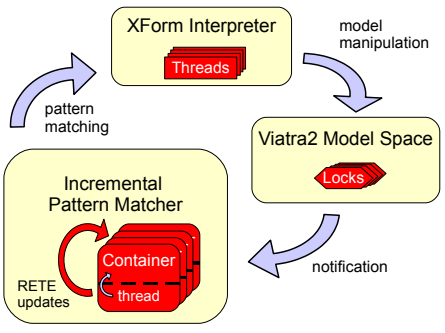


Figure 9: Architecture and workflow

VIATRA2 was designed with a modular architecture. In the context of this paper, the most important components are the model space, the transformation interpreter, and the incremental pattern matcher (see Figure 9). The model space contains and manages all modeling data, including models and metamodels. The transformation interpreter executes the ASM program that defines the transformation; during the process, pattern queries are directed towards the incremental pattern matcher and the model is potentially manipulated. Finally, pattern matcher modules are responsible for returning the results of pattern queries. The incremental pattern matcher accomplishes this by using pattern caches that are continously maintained in accordance to change notifications received from the model space and internal RETE updates.

[3] Environment: 2.2GHz Intel Core 2 Duo processor, Ubuntu 8.10, x64 OpenJDK 1.6.0_0, 1.5GB heap memory

The parallelization of graph transformation required changes in all three presented modules. The model space now offers a locking scheme to deal with conflicting transformation threads, as required by Subsection 4.3. The RETE net is divided into one or more containers, each operated by its dedicated thread, as described in Subsection 4.1 and Subsection 4.2. RETE updates can now either happen within one container, or be delivered as an asynchronous message between containers. Transformations are run by several interpreter threads that are executed in parallel, as proposed in Subsection 4.3.

5.2 Language features

VIATRA2 transformation designers have the option to select either a local search-based pattern matcher module or the incremental pattern matcher (see [BHRV09] for details). This annotation has been extended with a parameter to enable the parallelized pattern matching features described in Subsection 4.1 and Subsection 4.2.

The first listing in Figure 10 shows how this option was specified for the PNML generator example, selecting the parallelized version of the incremental pattern matcher for all pattern queries of the transformation *petrinet2PNML*. The annotation can also specify the type of pattern matcher on a per-pattern basis, thus the incremental and the local search-based pattern matchers can be combined. However, for performance considerations, it is not wise to mix the parallel and the non-parallel versions of the incremental pattern matcher, because it would cause two separate RETE nets to be built for the two techniques, which can result in a significant increase of memory consumption.

Multi-threaded model manipulation is also available from the language; the second listing in Figure 10 shows two ASM rule invocations (the two PNML generation jobs) executed in parallel, instead of sequentially, as proposed in Subsection 4.3.

```
@incremental('parallel'='1')
machine petrinet2PNML{
 ...
 rule generate(in Key, in PN) = seq {
  ...
 }
}
```

```
...
parallel
{
 call generate("out2.pnml", ...);
 call generate("out3.pnml", ...);
}
...
```

Figure 10: Language elements enabling parallel features

6 Related work

Incremental pattern matching. Incremental updating techniques have been widely used in different fields of computer science (including view updates in relational databases [GMS93]). In graph/model transformation tools, PROGRES [SWZ99] supports incremental attribute update performing immediate invalidation of partial matchings, while the validation of partial matchings are only computed on request (i.e., when a matching for the LHS is requested). The transformation engine of TefKat [LS05] performs an SLD resolution based interpretation to construct and incrementally maintain a search space tree representing the trace of transformation execution

[HLR06]. The uniform, incremental handling of model elements and patterns can be considered a unique, advanced feature of the approach. [VVS06] proposes to store a tree for the partial matches of a pattern, and incrementally updates it upon model changes.

RETE networks. RETE networks [For82], which stem from rule-based expert systems, have already been used as an incremental graph pattern matching technique in several application scenarios including the recognition of structures in images [BGT90], and the co-operative guidance of multiple uninhabited aerial vehicles in assistant systems as suggested by [MMS08]. Our contribution extends this approach by supporting a more expressive and complex pattern language.

Parallel RETE. There is also some work in literature in the context of parallel or distributed RETE implementations. For instance, [AT98] focuses on parallelizing rule applications, [MK90] parallelizes pattern matching. Unfortunately, certain approaches focusing on expert systems are hard to be accessed, e.g. due to vague patent descriptions [Lin05], and certain industrial solutions might not be published at all. Anyhow, these approaches rarely provide proofs to guarantee the global termination of local updates as mentioned in Subsection 4.2, which is specific to our model transformation context.

Parallel graph transformations. In addition to large amount of theoretical work on concurrent and parallel aspects of graph transformation, relatively little practical work has been carried out. Some advanced solutions were proposed by G. Mezei [Mez07] who analyses pattern conflicts and groups executable rules into *independence blocks* to execute them in parallel. Further contributions also introduced parallel pattern search for first occurence and all occurrences. Our current work is complementary to his work, as it offers parallelization with a different pattern matching paradigm. Future research shall be conducted to identify how to combine the strength of the two approaches.

The GrGen.NET transformation system follows a different approach [Sch08] to avoiding parallelization conflicts: the model itself is split into several partitions, and different threads operate on different fragments. Pattern matches bridging multiple partitions are dealt with separately.

As an alternative solution to managing conflicts, Roberto Bruni has recommendended that we should investigate the possibility of employing an optimistic concurrency strategy in the future.

7 Conclusion

The paper introduced various techniques for parallelizing graph transformation systems using incremental pattern matching. More specifically, we discussed how to exploit the power of modern computers with multiple processor cores, tailored to the specialities of incremental pattern matching. Our approach decouples model manipulation and pattern matching, and then parallelizes each of these phases.

We also sketched conditions when the proposed solutions are expected to perform best: (i) transformations with longer model manipulation sequences, (ii) transformation runs accessing different patterns, and (iii) transformation runs that are read-intensive.

Finally, an initial implementation of all the three presented ideas has been integrated to the VIATRA2 model transformation framework, and an initial performance evaluation of these parallelization techniques was carried out.

Acknowledgements: This work was partially supported by EU projects SENSORIA (IST-3-016004) and SecureChange (ICT-FET-231101), and the László Schnell Foundation.

Bibliography

[AT98] M. M. Aref, M. A. Tayyib. Lana—Match algorithm: a parallel version of the Rete—Match algorithm. *Parallel Comput.* 24(5-6):763–775, 1998. doi:http://dx.doi.org/10.1016/S0167-8191(98)00003-9

[Ber08] G. Bergmann. Incremental graph pattern matching and applications. Master's thesis, Budapest University of Technology and Economics, May 2008. http://home.mit.bme.hu/~bergmann/publications/bergmann_diploma_2008.pdf

[BGT90] H. Bunke, T. Glauser, T.-H. Tran. An Efficient Implementation of Graph Grammars Based on the RETE Matching Algorithm. In Ehrig et al. (eds.), *Graph-Grammars and Their Application to Computer Science*. Lecture Notes in Computer Science 532, pp. 174–189. Springer, 1990.

[BHRV08] G. Bergmann, A. Horváth, I. Ráth, D. Varró. A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation. In *ICGT*. 2008.

[BHRV09] G. Bergmann, A. Horváth, I. Ráth, D. Varró. Efficient Model Transformations by Combining Pattern Matching Strategies. In *Proc. of ICMT '09 , 2nd Intl. Conference on Model Transformation*. Springer, 2009. Accepted.

[BÖR+08] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, G. Varró. Incremental Pattern Matching in the VIATRA Model Transformation System. In Karsai and Taentzer (eds.), *Graph and Model Transformation (GraMoT 2008)*. ACM, 2008.

[BS03] E. Börger, R. Särk. *Abstract State Machines. A method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation*. Volume 2: Applications, Languages and Tools. World Scientific, 1999.

[For82] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19(1):17–37, September 1982.

[GMS93] A. Gupta, I. S. Mumick, V. S. Subrahmanian. Maintaining views incrementally (Extended abstract. In *In: Proc. of the International Conf. on Management of Data, ACM*. Pp. 157–166. 1993.

[Gra08] GraBaTs - Graph-Based Tools: The Contest. 2008. http://www.fots.ua.ac.be/events/grabats2008/.

[HLR06] D. Hearnden, M. Lawley, K. Raymond. Incremental Model Transformation for the Evolution of Model-Driven Systems. In Nierstrasz et al. (eds.), *MoDELS*. Lecture Notes in Computer Science 4199, pp. 321–335. Springer, 2006.

[JKW02] M. Jungel, E. Kindler, M. Weber. The Petri Net Markup Language. In *In S. Philipi, editor, Algorithmen und Werkzeuge fur Petrinetze (AWPN), Koblenz*. June 2002.

[Lin05] P. Lin. System and method to distribute reasoning and pattern matching in forward and backward chaining rule engines. US Patent application 20050246301, 02 2005.

[LS05] M. Lawley, J. Steel. Practical Declarative Model Transformation With Tefkat. In Bézivin et al. (eds.), *Proc. of the International Workshop on Model Transformation in Practice (MTiP 2005)*. October 2005. http://sosym.dcs.kcl.ac.uk/events/mtip05/.

[Mez07] G. Mezei. Supporting Transformation-Level Parallelism in Model Transformations. In *Automation and Applied Computer Science Workshop*. Budapest, Hungary, 2007.

[MK90] M. Mahajan, V. K. P. Kumar. Efficient parallel implementation of RETE pattern matching. *Comput. Syst. Sci. Eng.* 5(3):187–192, 1990.

[MMS08] A. Matzner, M. Minas, A. Schulte. Efficient Graph Matching with Application to Cognitive Automation. In Schürr et al. (eds.), *AGTIVE 2007*. Springer Verlag, 2008.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.

[Sch08] J. Schimmel. Parallelisierung von Graphersetzungssystemen. Master's thesis, Universitat Karlsruhe, 2008.

[SNZ08] A. Schürr, M. Nagl, A. Zündorf (eds.). *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers*. Lecture Notes in Computer Science 5088. Springer, 2008.

[SWZ99] A. Schürr, A. J. Winter, A. Zündorf. The PROGRES approach: language and environment. Pp. 487–550, 1999.

[VB07] D. Varró, A. Balogh. The model transformation language of the VIATRA2 framework. *Sci. Comput. Program.* 68(3):214–234, 2007.

[VVS06] G. Varró, D. Varró, A. Schürr. Incremental Graph Pattern Matching: Data Structures and Initial Experiments. In Karsai and Taentzer (eds.), *Graph and Model Transformation (GraMoT 2006)*. Electronic Communications of the EASST 4. EASST, 2006.